Load Balancing for Partition-based Similarity Search

Xun Tang, Maha Alabdulljalil, Xin Jin, Tao Yang
Department of Computer Science, University of California
Santa Barbara, CA 93106, USA
{xtang,maha,xin_jin,tyang}@cs.ucsb.edu

ABSTRACT

All pairs similarity search, used in many data mining and information retrieval applications, is a time consuming process. Although a partition-based approach accelerates this process by simplifying parallelism management and avoiding unnecessary I/O and comparison, it is still challenging to balance the computation load among parallel machines with a distributed architecture. This is mainly due to the variation in partition sizes and irregular dissimilarity relationship in large datasets. This paper presents a two-stage heuristic algorithm to improve the load balance and shorten the overall processing time. We analyze the optimality and competitiveness of the proposed algorithm and demonstrates its effectiveness using several datasets. We also describe a static partitioning algorithm to even out the partition sizes while detecting more dissimilar pairs. The evaluation results show that the proposed scheme outperforms a previously developed solution by up to 41% in the tested cases.

Categories and Subject Descriptors: H.3.3 [Information Search and Retrieval]: Clustering, Search Process

Keywords: All-pairs similarity search; load balancing; partitioning; competitiveness analysis

1. INTRODUCTION

All Pairs Similarity Search (APSS) [9], which identifies similar objects in a dataset, is used in many applications including collaborative filtering based on user interests or item similarity [1], search query suggestions [22], web mirrors and plagiarism recognition [23], coalition detection for advertisement frauds [20], spam detection [11, 17], clustering [7], and near duplicates detection [16]. The complexity of naïve APSS can be quadratic to the dataset size. Previous researches on expediting similarity computing developed filtering methods [9, 27, 4], inverted indexing [19, 21], or partitioning and parallelization techniques [3]. It is shown in [3] that APSS can be performed through a number of independent tasks where each compares a partition of vectors with other candidate vectors. This approach outperforms other approaches [19, 8] by an order of magnitude because of the simplified parallelism management and aggressive elimination of unnecessary I/O and comparison.

Given a large number of data partitions, we need to assign them to parallel machines and decide the direction of similarity comparison due to the symmetric property of comparison. Load imbalance can hugely affect scalability and overall performance. In this paper, we explore load balancing issues and develop a two-stage assignment algorithm to improve the execution efficiency of APSS. The first stage constructs a preliminary load assignment over tasks. The second stage refines the assignment to be more balanced. Our analysis shows that the developed solution is competitive to the optimum with a constant ratio. We further improve the dissimilarity detection ability in the static partitioning step, while producing partitions with relatively even sizes to facilitate the load balancing step.

The paper is organized as follows. Section 2 reviews some background and related work. Section 3 discusses the design framework. Section 4 introduces the two-stage load assignment algorithm and its performance properties. Section 5 presents the improved data partitioning scheme. Section 6 is the experimental evaluation. Section 7 concludes this paper and the appendix lists the proofs of the analytic results.

2. BACKGROUND AND RELATED WORK

Following the work in [9], the APSS problem is defined as follows. Given a set of vectors $d_i = \{w_{i,1}, w_{i,2}, \cdots, w_{i,m}\}$, where each vector contains at most m features and may be normalized to a unit length, the cosine-based similarity between two vectors is computed as:

$$Sim(d_i, d_j) = \sum_{t \in (d_i \cap d_j)} w_{\mathrm{i,t}} \times w_{\mathrm{j,t}}.$$

Two vectors d_i, d_j are considered similar if their similarity score exceeds a threshold τ, namely $Sim(d_i, d_j) \geq \tau$. The time complexity of APSS is high for a big dataset. There are application-specific methods applied for data preprocessing. For example, text mining removes stop-words or features with extremely high document vector frequency [7, 13, 19].

There are several groups of optimization techniques developed in the previous work to accelerate APSS.

Dynamic computation filtering. Partially accumulated similarity scores can be monitored at runtime and dissimilar document pairs can be detected dynamically without complete derivation of final similarity scores [9, 27, 21].

Similarity-based grouping in data pre-processing. The search scope for similarity can be reduced when potentially similar vectors are placed in one group. One can use an inverted index [27, 19, 21] developed for information retrieval [7]. This approach identifies vectors that share at least one feature as potentially similar, so certain data traversal is avoided. Similarly, the work in [25] maps feature-sharing vectors to the same group for group-wise parallel computation. This technique is more suitable for vectors with low sharing pattern, otherwise it suffers from excessive redundant computation among groups. Locality-sensitive hashing (LSH) can be considered as grouping similar vectors into one bucket with approximation [15, 24]. This approach has a trade-off between precision and recall, and may introduce redundant computation when multiple hash functions are used. A study [4] shows that exact comparison algorithms can deliver performance competitive to LSH when computation filtering is used. In partition-based APSS [3], dissimilar vectors are identified in the static partitioning step. The APSS problem is then converted to executing a set of independent tasks each compares one partition with some of the other partitions. These tasks can be executed in parallel with much simplified parallelism management.

Load balancing and scheduling. Exploiting parallel resources over thousands of machines for scalable performance is important and challenging. Load balancing is considered in the context of search systems for index serving [6, 18]. A recent study [26] introduces a division scheme to improve load balance for dense APSS problems using multiple rounds of MapReduce computation. In order to minimize the communication overhead while maintaining the computational load balance, in this paper, we focus on load balancing of APSS with record-based partitioning. The general load balancing and scheduling techniques for clusters and parallel systems have been extensively addressed in previous work. A simple greedy policy [14] that maps a ready task to a computation unit once it becomes idle is widely adopted (e.g. [10]). Scheduling for MapReduce systems such as Hadoop [12, 28] has followed the greedy policy to execute queued tasks on available cores and exploit data locality whenever feasible. Assuming that parallel tasks are scheduled following such a greedy policy, we address how these tasks should be formed considering scalability and efficiency.

3. FRAMEWORK

This section gives an overview of the partition-based framework [3] and presents the load balancing problem.

3.1 Partition-based Similarity Search

The framework for Partition-based Similarity Search (PSS) consists of two phases. The first phase divides the dataset into a set of partitions. During this process, the dissimilarity among partitions is identified so that unnecessary data I/O and comparisons among them are avoided. The second phase assigns a partition to each task at runtime and each task compares this partition with other potentially similar partitions. These tasks are independent when running on a set of parallel machines. Figure 1 depicts the whole process.

Dissimilarity-based partitioning identifies dissimilar vectors without explicitly computing the product of their features. One approach [3] utilizes the following inequality that calculates the 1-norm and ∞-norm of each vector:

$$Sim(d_i, d_j) \leq min(||d_i||_\infty ||d_j||_1, ||d_j||_\infty ||d_i||_1) < \tau.$$

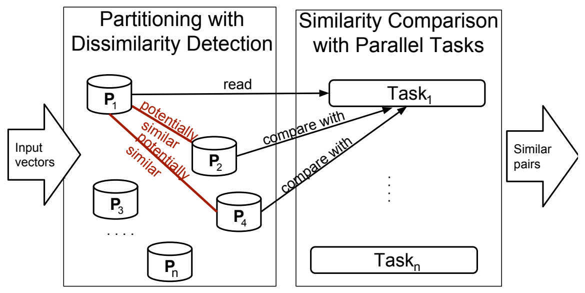


Figure 1: Illustration of partition-based similarity search.

```
Task $T_k$
  Read all vectors from assigned partition $P_k$
  Build inverted index of these vectors
  Conduct self-comparison among vectors in $P_k$
  repeat
    Fetch a potentially similar partition
    for $d_j \in$ fetched partition do
      Compare ($P_k$, $d_j$)
  until all non-dissimilar partitions are fetched
```

Figure 2: Definition of each PSS task.

The partitioning algorithm sorts the vectors based on their 1-norm values first. It then uses the sorted list to identify dissimilar pairs (d_i, d_j) satisfying inequality $\|d_i\|_1 < \frac{\tau}{||d_j||_\infty}$. A different τ value would affect the outcome of the dissimilarity-based partitioning.

Once the dataset is separated into v partitions, v independent tasks are scheduled. Each task is responsible for a partition and compares this partition with all potentially similar partitions. We assume that the assigned partition for each task fits the memory of one machine as the data partitioning can be adjusted to satisfy such condition. Other partitions to be compared with may not fit the remaining memory and need to be fetched gradually from a local or remote storage. In a computing cluster with a distributed file system such as Hadoop, tasks can seamlessly fetch data without concerning about the physical locations of data.

Figure 2 describes the function of each task T_k in partition-based similarity search. Task T_k loads the assigned partition P_k and produces an inverted index to be used during the partition-wise comparison. Next, T_k fetches a number of vectors from potentially similar partitions and compares them with the local partition P_k. Fetch and comparison is repeated until all candidate partitions are processed.

3.2 Load Assignment Problem

We formalize the load assignment problem as follows. The data partitioning phase defines a set of v partitions and their potentially similar relationship. This can be represented as a graph, called a similarity graph defined next.

Definition 1 Similarity graph (G): *Let G be an undirected graph where each node represents a data partition and each edge indicates potential similarity relationship between the two partitions it connects.*

Since the similarity result of two vectors is symmetric, comparison between two partitions P_i and P_j should be only conducted by one of the corresponding tasks T_i or T_j. A load assignment algorithm determines which task performs this comparison. The load assignment process converts the undirected similarity graph into a directed graph in which the direction of each edge indicates which task conducts the corresponding comparison. We call this a comparison graph and it is defined as follows.

Definition 2 Comparison graph (D): *Let D be a directed graph where each node represents a data partition. An edge $e_{i,j}$ from partition P_i to P_j indicates that task T_j compares P_j with P_i.*

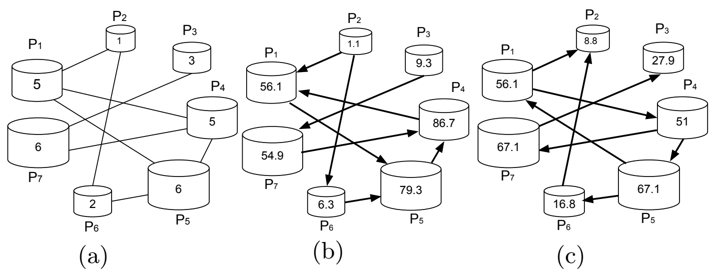


Figure 3: (a) An undirected similarity graph; node weights are partition sizes. (b) A directed comparison graph for (a); node weights are the corresponding task cost. (c) Another comparison graph for (a).

Comparison graph D contains the same set of nodes and edges as the corresponding similarity graph G, except that the edges in D are directed. The directed edges reveal the data flow direction when comparing two potentially similar partitions. Figure 3(a) illustrates a similarity graph with seven nodes. P_1 is potentially similar to P_2, P_4 and P_5, for instance. The comparison between P_1 and P_2 can be performed by either T_1 or T_2. The numbers marked inside the graph nodes are partition sizes, proportional to the number of vectors in the partition. Figures 3(b) and 3(c) show two comparison graphs with different load assignments. The number marked inside a comparison graph node is the corresponding task cost and we explain the cost model below.

The cost function of each task consists of computation cost and data I/O cost. For each task defined in Figure 2, the computation cost includes the cost of an inverted index look-up, multiplication and addition, and memory/cache accesses. While a thorough cost model involves memory hierarchy analysis [2], the overall computation cost can be approximated as proportional to the size of the corresponding partition P_i multiplied by the size of the potentially similar partitions to be compared with. The data I/O cost occurs when fetching P_i and other partitions from local or remote machines, and also when storing the detected similarity results on disk. Since the start-up I/O cost and transmission bandwidth difference to the local or remote storage are relatively small, the I/O cost is approximately proportional to the size of the partitions involved. Note that the runtime scheduling that maps tasks to machines is affected by data locality. As we discuss later, the computation cost is dominating in APSS and thus the I/O cost difference caused by data locality is not sufficient enough to alter our optimization results in terms of competitiveness to the optimum.

Define the cost of task T_i corresponding to partition P_i in comparison graph D as:

$$Cost(T_i) = f(P_i, P_i) + f_c(P_i) + \sum_{e_{j,i} \in D} (f(P_i, P_j) + f_c(P_j))$$

where $f(P_i, P_i)$ is the self comparison cost for partition i and is quadratically proportional to the size of P_i. $f(P_i, P_j)$ is the comparison cost between partition i and j. It satisfies that $f(P_i, P_j) = f(P_j, P_i)$ and this cost is proportional to the size of P_i multiplied by size of P_j. $f_c(P_i)$ is the I/O and communication cost to fetch partition P_i from local and/or remote storage and output the results of self-comparison. $f_c(P_j)$ is the cost to fetch partition P_j and output the similar pairs between P_i and P_j. For Figures 3(b) and 3(c), $f(P_i, P_j)$ is a multiplication of the sizes of P_i and P_j, and $f_c(P_i)$ is estimated as 10% of the size of P_i. In Figure 3(c), $Cost(T_5)$=67.1 because $f(P_5, P_5)$=36, $f(P_5, P_4)$=30, $f_c(P_5)$=0.6 and $f_c(P_4)$=0.5.

Different edge direction assignments can lead to a large variation in task weights. Let $Cost(D) = \max_{P_i \in D} Cost(T_i)$. For example, in Figure 3(b) $Cost(D)$=86.7 based on $Cost(T_4)$. In Figure 3(c) $Cost(D)$=67.1. Deriving a comparison graph that minimizes the maximum cost among all tasks is a key strategy in our design. As the load is shifted from the heaviest task to the other tasks, better load balancing is achieved.

A circular mapping solution in [3] compares a partition with half of other partitions, if they are potentially similar. When the number of partitions is odd, task T_i compares P_i with partitions P_j where j belongs to the set: $i\%v + 1,\ (i+1)\%v + 1,\ \cdots,\ (i + \frac{v-3}{2})\%v + 1$. Figure 3(b) shows the circular solution for the similarity graph in Figure 3(a). T_1 is assigned to compare with partitions from P_2 to P_4, hence the edge is directed from P_2 and P_4 to P_1. Similarly, the comparison between P_1 and P_5 is assigned to P_5. The circular approach is reasonable when the distribution of node connectivity and partition sizes is not skewed. In practice, that is often not true.

Dataset	Partition size (# of records per partition)			Task cost
	Avg	Std. Dev/Avg	Max/Avg	Max/Avg
Twitter	143,042	1.75	6.85	2.14
ClueWeb	337,720	0.67	2.37	4.25
YMusic	21,550	0.82	4.35	8.97

Table 1: Distribution statistics for partition size and parallel execution time with circular load assignment.

Table 1 shows the variance of partition sizes and task costs in three datasets. The largest partition size could be many times larger than the average partition size and the standard deviation compared to the average size is also high. Additionally, the similarity relationship among partitions is highly irregular. Some partitions have lots of edges in similarity graph while others have sparse connections. Circular load assignment treats all partitions equally regardless of such variations and as a result, a task could be assigned all the comparison loads while its counterpart tasks are very light. Column 5 of Table 1 shows the maximum divided by average task cost using circular assignment.

The ultimate goal of load assignment is to schedule computation to parallel machines with minimum job completion time. Since undirected edges in a similarity graph creates uncertainty in task workload, the key question here is what to optimize. Will balancing the task costs computed from the comparison graph help speedup the runtime execution without knowing the allocated computing resource in advance? In the next section, we discuss our optimization strategy and present a two-stage assignment algorithm.

4. LOAD BALANCING OPTIMIZATION

Our algorithm for load assignment consists of two stages to derive a comparison graph with balanced load among tasks. The design considers uneven partition sizes and irregular dissimilarity relationship. The derived tasks are scheduled at runtime to q cores and the tasks with reduced variation in sizes contribute to better performance after scheduling. We will show that such a strategy can produce a solution competitive to the optimal solution for scheduling a similarity graph on a given number of cores. We discuss the two-stage algorithm in the following two subsections.

4.1 Stage 1: Initial Load Assignment

The purpose of Stage 1 of this algorithm is to produce an initial load assignment such that tasks with small partitions conduct more comparisons. This stage performs v steps where v is the total number of partitions in the given similarity graph. Each step identifies a partition, determines the direction of its similarity edges, and adds this partition along with these directed edges to comparison graph.

More specifically, each step works on a subgraph of the original undirected graph G, called G_k at step k. G_1 is the original graph G. At step k, the algorithm identifies partition P_x with the lowest *potential computation weight* (PW). The potential computation weight for task T_x based on subgraph G_k is defined as:

$$PW(G_k, P_x) = f(P_x, P_x) + \sum_{e_{x,y} \in G_k} f(P_x, P_y).$$

It represents the largest possible computation weight for task T_x given the undirected edges in G_k. G_{k+1} is derived from G_k by removing the selected partition P_x and its edges in G_k. These edges connecting P_x in G_k are chosen to point to P_x in the generated directed graph.

Node	Init	Step 1	Step 2
	G_1	G_2	G_3
P_1	85	80	80
P_2	8	-	-
P_3	37	37	37
P_4	110	110	110
P_5	108	108	96
P_6	18	16	-
P_7	84	84	84

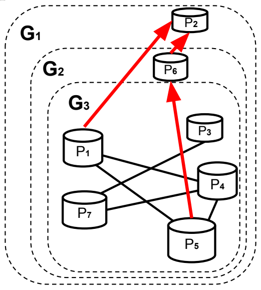


Figure 4: The first two steps in Stage 1 in the right figure, along with the PW values in the left table.

Figure 4 illustrates the first two steps in Stage 1. The left part of the figure lists the initial PW values of each node, as well as the corresponding values after the first step and second step. Partition P_2 has the lowest PW value initially and is selected at Step 1. Edges connecting P_2 are all directed to P_2 in the formed directed graph. The PW values of the partitions adjacent to P_2 are changed from G_1 to G_2. Step 2 identifies P_6 as the the lowest PW in G_2, removing it and its edges from G_2. Finally the outcome of Stage 1 produces a comparison graph shown in Figure 5(a).

The cost of a task at Step k is considered to be *determined* if its corresponding partition has been selected before Step k. Otherwise, a task has a *potential* cost that equals to PW value plus possible I/O cost. Figure 6 shows the standard deviation of task costs at the first 200 steps using $Cost(T_i)$ if this task is determined, or its potential computation weight

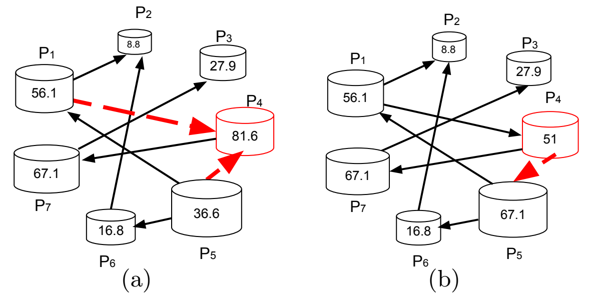


Figure 5: (a) The assignment produced in Stage 1. (b) The first refinement step in Stage 2: reversing edge $e_{5,4}$ to $e_{4,5}$.

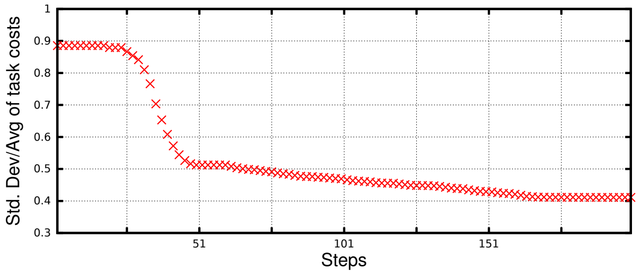


Figure 6: Monotonic decrease of the cost standard deviation in the first 200 steps in Stage 1 for Twitter dataset. The values are normalized by the average task computation cost.

PW if it is undetermined. The step-wise trend illustrates that Stage 1 gradually reduces the variation of task costs.

Stage 1 pushes the computation load to the tasks with potentially low weight. This technique works better when partitions have highly skewed sizes since the lightest partitions absorb as much workload as possible. However, this greedy heuristic may cause some tasks to carry an excessive amount of computation. Another issue is that Stage 1 does not consider data I/O and communication cost, so the effect of optimization might be weakened. Hence, we introduce Stage 2 to further refine the assignment produced by Stage 1 and mitigate the aforementioned weakness.

4.2 Stage 2: Assignment Refinement

Stage 2 conducts a number of refinement steps to reduce the load of the heavy tasks by gradually shifting part of their computation to their lightest neighbors. It performs the following procedure:

1. Find the task with the highest assigned cost $Cost(T_x)$. Identify one of P_x's incoming neighbors, say P_y, with the lowest cost among these neighbors, and reverse the direction of this edge from $e_{y,x}$ to $e_{x,y}$. Such a reversion causes a cost increase for T_y and a cost decrease for T_x. However, if the new cost of T_y becomes the same or larger than the original cost of T_x, this edge reversion is rejected. When an edge reversion is rejected, we continue with the incoming neighbor that has the second lowest cost. Repeat this process until a suitable neighbor is found so that the edge reversion successfully reduces $Cost(T_x)$. If all incoming neighbors of P_x are probed but no flip reduces $Cost(T_x)$ successfully, mark $Cost(T_x)$ as non-reducible.
2. Repeat the above step for the task with the highest weight after the update. If such a task is non-reducible, try the reducible task with the next highest weight. If all nodes are marked non-reducible or the number of iterations tried reaches a predefined limit, the algorithm stops.

Figure 5(b) depicts the first refinement upon the output of Stage 1. The first edge probed in Figure 5(a) is $e_{5,4}$ because T_4 has the highest cost and T_5 has the lowest cost among all incoming neighbors of P_4 (i.e. P_1 and P_5). The reversion of edge $e_{5,4}$ to $e_{4,5}$ reduces $Cost(T_4)$ from 81.6 to 51 and boosts T_5 to be the task with the highest assigned weight, ready for the next probe. Since the flip of any incoming edge to P_5 does not further reduce $Cost(T_5)$, we do not flip. Finally, Stage 2 produces a comparison graph as shown in Figure 3(c) with $Cost(D)$=67.1.

4.3 Competitiveness Analysis

We do not know how the optimum scheduling solution dynamically maps tasks to machines at runtime as shown in Figure 7. However, we can use a bound analysis to show that our heuristic approach performs competitively in a constant factor compared to the optimum. We first address the load balancing issue without awareness of the machine location. Network distances impact the I/O and communication cost, but this cost is relatively less significant compared to computation load imbalance in PSS. Define

$$\delta = \max_{P_i \in G}\left(\frac{f_c(P_i)}{f(P_i, P_i)}, \max_{e_{j,i} \in G} \frac{f_c(P_j)}{f(P_i, P_j)}\right).$$

This ratio represents the overhead ratio of I/O and communication involved in each task compared to its computation. In our experiments as shown in Table 3, I/O overhead is relatively small. Given this computation-dominating setting, for a cluster of machines with multiple CPU cores, we will simply view that the whole cluster has q cores without differentiating their machine location. The overhead in accessing data locally or remotely is captured in ratio δ.

Theorem 1 shows the result of two-stage load assignment algorithm is competitive to the smallest possible cost without knowing the number of cores available. Theorems 2 and 3 characterize the competitiveness of the algorithm to the optimum when the similarity graph is scheduled to q cores. The theorem proofs are listed in the appendix.

Theorem 1 *Define $Cost_{min}(G)$ as the smallest cost of a comparison graph derived from a given similarity graph G. The two-stage load assignment algorithm produces a comparison graph D with $Cost(D)$ competitive to $Cost_{min}(G)$. Their relative ratio satisfies*

$$Cost(D) \le 2(1+\delta)Cost_{min}(G).$$

The above result shows that the tasks produced by the two-stage algorithm have a fairly balanced cost distribution. As illustrated in Figure 7, a simple runtime scheduling heuristic is to assign tasks to idle computing units whenever they become available [14]. For example, the Hadoop MapReduce [12] scheduler works by assigning ready tasks in a greedy fashion with the best effort of preserving data locality. Once the central job tracker detects the availability of a task tracker, it assigns a ready task to the task tracker as long as there exists an unassigned task. When deciding which task to assign, it favors the tasks processing data local to or close to the machine of the task tracker. What is the performance behavior of our comparison tasks scheduled under such a greedy policy?

The next theorem shows that under a greedy scheduler, the tasks produced by the two-stage algorithm perform competitively compared to an optimum solution.

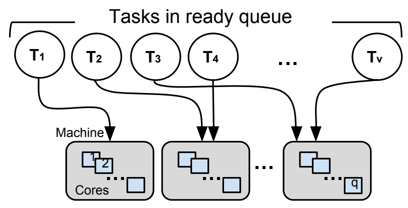


Figure 7: Greedy execution of v tasks at runtime on a cluster of machines with q cores.

Theorem 2 *The two-stage load assignment with a greedy scheduler produces a solution with job completion time PT_q competitive to the optimal solution with completion time PT_{opt}. Their relative ratio for dedicated q cores satisfies*

$$\frac{PT_q}{PT_{opt}} \le (3 - \frac{2}{q})(1+\delta).$$

Our analysis in the appendix shows that with computation-dominating tasks and a greedy scheduling policy, the upper bound of execution time is affected by the weight of the heaviest task. This supports our load balancing optimization that targets the minimization of the maximum task weight during load assignment.

Stage 1 may produce an unbalanced initial assignment in which some nodes absorb too much computation, especially in dense graphs. Stage 2 mitigates this issue with a sequence of refinements. The following theorem illustrates that for a fully connected graph, our approach delivers a near-optimal solution, and it can be inferred from the proof that the refinement process carried out in Stage 2 is the main reason that this goal is accomplished.

Theorem 3 *The two-stage load assignment with a greedy scheduler is competitive to the optimum for a fully connected similarity graph with equal partition sizes and equal computation costs in self-comparison and inter-partition comparison. Their relative ratio satisfies*

$$\frac{PT_q}{PT_{opt}} \le 1+\delta.$$

5. DATA PARTITIONING OPTIMIZATION

This section presents an improved partitioning method for Phase 1 of partition-based similarity search presented in [3]. The goal of this improvement is twofold: 1) to detect more dissimilarity among partitions to avoid unnecessary data I/O and comparison, and 2) to reduce the size gap among partitions and facilitate the load balancing process.

5.1 Dissimilarity Detection with Hölder's Inequality

To identify more dissimilar vectors without explicitly computing the product of their features, we use Hölder's inequality to bound the similarity of two vectors:

$$Sim(d_i, d_j) \le \|d_i\|_r \|d_j\|_s$$

where $\frac{1}{r} + \frac{1}{s} = 1$. $\|\cdot\|_r$ and $\|\cdot\|_s$ are r-norm and s-norm values. r-norm is defined as

$$\|d_i\|_r = (\sum_t |w_{i,t}|^r)^{1/r}.$$

With $r = 1$, $s = \infty$, the inequality becomes $Sim(d_i, d_j) \le \|d_i\|_1 \|d_j\|_\infty$, which is a special case introduced in [3].

If the similarity upper-bound is less than τ, such vectors are not similar and comparison between them can be avoided. The algorithm that produces partitions following Hölder's inequality is described as follows.

1. Divide all vectors evenly to produce l consecutive layers $L_1, L_2, \cdots, L_l$ such that all vectors in L_k have lower r-norm values than the ones in L_{k+1}.
2. Subdivide each layer further as follows. For the i-th layer L_i, divide its vectors into i disjoint sublayers $L_{i,1}$, $L_{i,2}$, $\cdots$, $L_{i,j}$. With $j < i$, members in sublayer $L_{i,j}$ are extracted from L_i by comparing with the maximum r-norm value in layer L_j:

$$L_{i,j} = \{d_x | d_x \in L_i \text{ and } \max_{d_y \in L_j} \|d_y\|_r < \frac{\tau}{\|d_x\|_s}\}.$$

This partitioning algorithm has a complexity of $O(n \log n)$ for n vectors and can be easily parallelized. Each sublayer is considered as a data partition and these partitions have dissimilarity relationship with the following property.

Proposition 1 *Given $i > j$, vectors in sublayer $L_{i,j}$ are not similar to the ones in any sublayer $L_{k,h}$ where $k \leq j$ and $k \geq h$.*

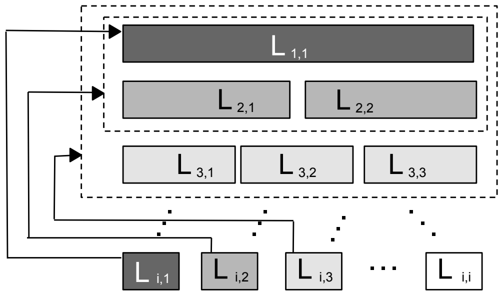


Figure 8: Dissimilarity relationship among data partitions.

Figure 8 illustrates the dissimilarity relationship among these sublayers as partitions and each pointing edge represents a dissimilarity relationship. For example, $L_{i,2}$ is not similar to $L_{1,1}, L_{2,1}$, or $L_{2,2}$ in the top two layers.

5.2 Even Partition Sizes

To facilitate load balancing in the later phase, we aim at creating more evenly-sized partitions at the dissimilarity detection phase. One way is to divide the large sublayers into smaller partitions. Its weakness is that it introduces more potential similarity edges among these partitions, hence the similarity graph produced becomes denser, more communication and I/O overhead are incurred during runtime. Another method targets at approximately the same $L_{i,j}$ size for any $i \leq j$ using a non-uniform layer size. For example, let the size of layer L_k be proportional to the index value k, following the fact that the number of sublayers in L_k is k in our algorithm. The main weakness of this approach is that less dissimilarity relationships are detected as the top layers become much smaller.

We adopt a hierarchical partitioning that identifies large sublayers, detects dissimilar vectors inside these sublayers, and recursively divides them using the procedure discussed in Section 5.1. The recursion stops for a sublayer when reaching a partition size threshold. Each partition inherits the dissimilar relationship from its original sublayer. The new partitions together with the undivided sublayers form the undirected similarity graph G ready for load assignment.

6. PERFORMANCE EVALUATION

6.1 Implementation Details

We have implemented our algorithms in Java using Hadoop MapReduce. Prior to the comparison computation, records are grouped into dissimilar partitions and this partitioning step including norm value sorting is parallelized. The cost of parallel partitioning is relatively small and is roughly 3% of the total parallel execution time in our experiments. During the load balancing step, the two-stage algorithm defines the comparison direction among potentially similar partitions, generates a comparison graph stored in a distributed cache provided by Hadoop, and derives a set of parallel tasks defined in Figure 2.

Hadoop runtime scheduler monitors the load of live nodes in the cluster and assigns a PSS task to the first idle core. Such a dynamic and greedy scheme can absorb potential skewness in data that fluctuates the actual computational cost. Theorem 2 reflects the competitiveness of PSS tasks scheduled under Hadoop greedy policy. During execution, each task loads the assigned partition with a user-defined reader, obtains a list of partitions to be compared with from the comparison graph file, and loops through the partition list to conduct partition-wise comparison.

6.2 Datasets and Metrics

The following datasets are used: 1) Twitter dataset containing 100 million tweets with 18.5 features per tweet on average after pre-processing. Dataset includes 20 million real user tweets and additional 80 million synthetic data generated based on the distribution pattern of the real Twitter data but with different dictionary words. 2) ClueWeb dataset containing about 40 million web pages, randomly selected from the ClueWeb collection [5]. The average number of features is 320 per web page. We choose 40M records because it is already big enough to illustrate the scalability. 3) Yahoo! music dataset (YMusic) used to investigate the song similarity for music recommendation. It contains 1,000,990 users rating 624,961 songs with an average feature vector size 404.5.

Dataset	Twitter		ClueWeb		YMusic
Size	4M	100M	1M	40M	625K
AMD	45	45,157*	50	79,845*	31.95
Intel	26.7	25,438*	29.3	46,946*	17.8
AMD/df-limit	1.27	797*	4.55	7,286*	6.23

Table 2: Sequential time in hours on AMD Opteron 2218 2.6GHz and Intel X5650 2.66GHz processors (τ=0.8).

Experiments are conducted on a cluster of servers each with 4-core AMD Opteron 2218 2.6GHz processors and 8G memory and a cluster with Intel X5650 6-core 2.66GHz dual processors and 24GB of memory per node. Rows 3 and 4 of Table 2 list the sequential execution time in hours for three benchmarks with different input sizes when running PSS with static partitioning and two-stage load balancing. We set similarity threshold τ as 0.8 throughout our experiments unless otherwise specified. The values marked with * are estimated by sampling part of its computation tasks and considering the fact that computation load grows quadratically as problem size grows. From the results in Rows 3 and 4, APSS is a time consuming process. Even for a Twitter dataset with 20M tweets, the entire dataset can fit in the memory; but it still takes many days to produce the results. Parallelization can shorten the job turnaround time and speedup iterative data analysis and experimentation.

Stop words are removed in the Twitter and ClueWeb input datasets; additional approximated preprocessing may be applied to reduce sequential time significantly if the trade-off in accuracy is acceptable [13, 19]. For example, the bottom row of Table 2, marked as "df-limit", lists the sequential time on an AMD core after removing features with their vector frequency exceeding an upper limit proposed in [19]. After sampling a 8M ClueWeb dataset, 49 words with document frequency above 200,000 are excluded in web page comparison and the sequential time is shortened by 11x. Using this df-limit strategy reduces the sequential time by 35.3x or more for Twitter and by 5.1x for YMusic. In the rest of this section, we report performance without using approximated preprocessing such as df-limit.

Noted that the algorithms discussed in this paper conduct *exact* similarity comparison with no approximation for a given input dataset. One may consider using approximation such as LSH mapping which roughly groups similar vectors into buckets and our algorithm can be applied within each big bucket produced by LSH. This is beyond the scope of this paper and we will investigate this in the future work.

In the rest of this section, we mainly report performance on the AMD cluster because a larger Intel cluster environment was less available for us to conduct experiments. Our evaluation has the following objectives: 1) Demonstrate the problem complexity and the execution scalability of tasks produced by the two-stage load balancing method. We report the speedup over the sequential time as we scale the number of cores. 2) Assess the effectiveness of the proposed two-stage optimization compared with the circular load balancing scheme. 3) Evaluate the performance of the generalized static partitioning algorithm in detecting dissimilarity and narrowing the size gaps among partitions.

6.3 Scalability and Comparisons

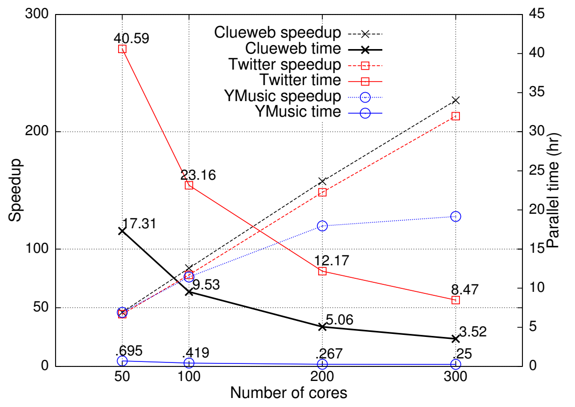


Figure 9: X axis is the number of cores used. Left Y axis is speedup. Right Y axis is parallel execution time.

Figure 9 shows the speedup and parallel time for processing 40M ClueWeb dataset, 100M Twitter dataset and 625K YMusic dataset when varying the number of AMD cores. Due to the time constraint in our shared cluster environment, we report the average execution time of multiple runs after randomly selecting 10% of ClueWeb parallel tasks and 20% of Twitter tasks. Such a sampling methodology follows the one used in [19]. Speedup is defined as the sequential time of these tasks divided by the parallel time. The performance of our scheme scales well as the number of CPU cores increases. The efficiency is defined as the speedup divided by the number of cores used. For the two larger datasets, the efficiency is about 83.7% for ClueWeb and 78% for Twitter when 100 cores are used. When running on 300 cores, the efficiency can still reach 75.6% for ClueWeb and 71.7% for Twitter. The decline is most likely caused by the increased I/O and communication overhead among machines in a larger cluster. Efficiency for YMusic with 31.95 hour sequential time are 76.2% with 100 cores and 42.6% with 300 cores. There is no significant reduction of parallel time from 200 cores to 300 cores, remaining about 15 minutes. The problem size of this dataset is not large enough to use more cores for amortizing overhead. Still parallelization shortens search time and that can be important for iterative search experimentation and refinement.

We also calculate the average time for comparing each pair of vectors normalized by their average length in a dataset. Namely $\frac{\text{Parallel time} \times \text{No of cores}}{\text{No of pairs} \times \text{Vector length}}$. The normalized pair-wise comparison time is about 1.24 nanoseconds for Twitter and 0.74 nanoseconds for ClueWeb using 300 AMD cores given $\tau = 0.8$. Varying the number of cores affects due to the difference in parallel efficiency. Varying τ also affects because it changes the results of dissimilarity-based partitioning and graph structure. This number can become smaller if approximated preprocessing is adopted [13, 19].

To confirm the choice of partition-based search, we have also implemented an alternative MapReduce solution to exploit parallel score accumulation following the work of [19, 8] where each mapper computes partial scores and distributes them to reducers for score merging. The parallel score accumulation is much slower because of the communication overhead incurred in exploiting accumulation parallelism. For example, to process 4M Twitter data using 120 cores, parallel score accumulation is 19.7x slower than partition-based similarity search which has much simpler parallelism management and has no shuffling between mappers and reducers. To process 7M Twitter data, parallel score accumulation is 25x slower. As a sanity check, we also estimate the normalized pair-wise comparison times reported in [19]. To compare 90K vectors with 4.59 million MEDLINE abstracts using at most 60 terms per vector on about 120 cores each with 2.8GHz CPU, it takes a MapReduce solution called PQ [19] 448 minutes with approximated preprocessing, meaning about 130.1 nanoseconds to compare each normalized vector pair.

Dataset	Cores	Static Partitioning	Similarity Comparison		
			Read	Write	CPU
Twitter	100	2.8%	0.9%	11.7%	84.6%
ClueWeb	300	2.1%	1.9%	7.8%	88.2%
YMusic	20	3.0%	2.3%	1.8%	92.9%

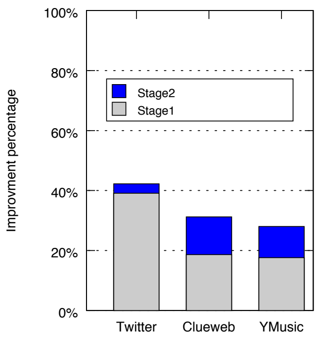
Table 3: Cost of static partitioning and runtime cost distribution of PSS in parallel execution.

Table 3 shows that static partitioning which is also parallelized takes 2.1% to 3% of the total parallel execution time.

The table also shows the time distribution in terms of data I/O and CPU usage for similarity comparison. Data I/O is to fetch data and write similarity results in the Hadoop distributed file system. This implies that the computation cost in APSS is dominating and hence load balance of the computation among cores is critical for overall performance.

6.4 Effectiveness of Two-Stage Load Balance

The experiments discussed in the previous sub-section have adopted the two-stage load balancing and improved static partitioning. In the rest of this section, we assess the impact of optimization using 100 AMD cores for 20M Twitter, 300 cores for 8M ClueWeb, and 20 cores for YMusic. We choose these sizes for faster experimentation while the performance impact of optimization for larger sizes is similar.



(a)

	Circular	2-stage
Max/Avg		
Twitter	2.14	1.45
ClueWeb	4.25	3.25
YMusic	8.97	6.68
Std. Dev/Avg		
Twitter	1.44	0.83
ClueWeb	2.68	1.77
YMusic	2.87	2.06

(b)

Figure 10: (a) Parallel time reduction contributed by Stages 1 and 2 compared to the circular assignment. (b) Maximum task cost and standard deviation over the average task cost with circular assignment or with two-stage assignment.

Figure 10(a) shows the improvement percentage in parallel time using two-stage load assignment compared to the baseline circular assignment. Parallel time with two-stage assignment is about 23.2 hours for Twitter, 14 hours for ClueWeb, and 1.7 hours for YMusic respectively. The figure also marks the improvement percentage contributed by Stage 1 and Stage 2 respectively. The overall improvement from the two-stage load assignment is 41% for Twitter, 32% for ClueWeb, and 27% for YMusic. Stage 1 contributes a large portion of the total improvement. Stage 2 contributes about 4% for Twitter, 12% in ClueWeb, and 10% for YMusic. Similarity graphs of ClueWeb and YMusic are denser and Stage 1 can be too aggressive in making the light partitions absorb too much comparison computation. Hence, the refinements in Stage 2 become more effective in such cases.

To examine the weight difference across all tasks, Figure 10(b) shows the maximal task weight with circular mapping or with the two-stage balancing method divided by the average task cost. It also lists the cost standard deviation divided by the average task cost. The larger these two ratios are, the more severe load imbalance is. Compared to circular mapping, the two-stage assignment reduces the Max./Avg. ratio by 32.2%, 23.5%, and 25.5% for Twitter, ClueWeb, and YMusic datasets respectively. For Std. Dev./Avg. ratio, the reduction is 42.4%, 34.0%, and 28.2% respectively.

6.5 Improved Data Partitioning

Figure 11 provides a comparison of the improved data partitioning with different r-norms. Y axis is the percentage of pairs detected as dissimilar. r=1 reflects the results in [3]. For ClueWeb, 19% of the total pairs under comparison are detected as dissimilar with r=3 while only 10% for r=1. For Twitter, the percentage of pairs detected as dissimilar is 34% for r=4 compared to 17% for r=1. The results show that choosing r as 3 or 4 is most effective. We have used the best r value for partitioning each dataset.

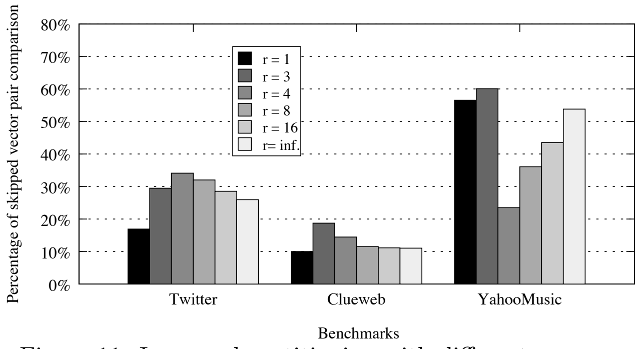


Figure 11: Improved partitioning with different r-norms.

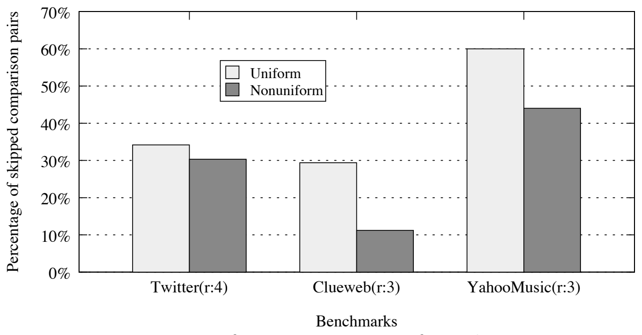


Figure 12: Uniform v.s. non-uniform layer size.

As discussed in Section 5.2, the initial layer size selection affects the size variation of the final partitions. Figure 12 gives a comparison of using uniform layer size and using non-uniform size with the marked r-norm settings. The uniform-sized layers yields better results. For ClueWeb, the uniform layers detect 2.6x as many dissimilar pairs compared to the non-uniform layers. Thus we opt for the uniform layers and recursively apply hierarchical partitioning to even out the sizes of sublayers.

Table 4 shows the effectiveness of recursive hierarchical data partitioning. The ratio of standard deviation of partition sizes over the average size drops by 9.7% for Twitter, 22.3% for ClueWeb, and 3.7% for YMusic. The relatively even workload benefits the task load balancing process and reduces parallel execution time by 5% to 18% additionally.

Dataset	Std. Dev/Avg (Without)	Std. Dev/Avg (With)	Parallel time reduction
Twitter	1.75	1.58	8.23%
ClueWeb	0.67	0.52	18.23%
YMusic	0.82	0.79	5.29%

Table 4: Change of partition sizes and parallel time with or without the recursive hierarchical partitioning.

7. CONCLUDING REMARKS

The main contribution of this paper is a two-stage load balancing algorithm for efficiently executing partition-based similarity search in parallel. The analysis provided shows its competitiveness to the optimal solution. This paper also presents an improved and hierarchical static data partitioning method to detect dissimilarity and even out the partitions sizes. Our experiments demonstrate that the two-stage load assignment improves the circular assignment by up to 41% in the tested datasets. The improved static partitioning avoids more unnecessary I/O and communication and reduces the size gaps among partitions with up to 18.23% end-performance gain in the tested cases.

Static partitioning can be processed efficiently in parallel, and could be further extended to handle incremental updates. Another future work is to study a hybrid scheme that integrates approximate methods such as LSH with our exact method for larger datasets when a trade-off between speed and accuracy is acceptable.

Acknowledgments. We thank Paul Weakliem, Kadir Diri, and Fuzzy Rogers for their help with the computer cluster, and Xifeng Yan, Ömer Eğecioğlu, and the anonymous referees for their insightful comments. This work is supported in part by NSF IIS-1118106 and Kuwait University Scholarship. Equipment access is supported in part by the Center for Scientific Computing at CNSI/MRL under NSF DMR-1121053 and CNS-0960316. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

APPENDIX

Theorem 1

PROOF. Let $Cost_1(D)$ be the value of $Cost(D)$ after Stage 1. Refinements in Stage 2 do not increase $Cost(D)$ and thus $Cost(D) \leq Cost_1(D)$. We just need to show that Stage 1 can reach a solution competitive to $Cost_{min}(G)$. Namely $Cost_1(D) \leq 2(1+\delta)Cost_{min}(G)$.

Let D_i be a directed graph with all nodes $\in G_i$ and all edge orientations determined through the steps from G_i to G_{v-1} in stage 1, given a total of v partitions and $D_1{=}D$, $G_1{=}G$.

We use an induction to prove this theorem. The induction goes from D_{v-1} to D_1, reversing to the creation process in Stage 1. Towards the end of Stage 1, subgraph G_{v-1} has two nodes left, and at most one edge between them. Choosing the partition with the smaller computation weight to perform the inter-partition comparison will add some communication and I/O cost, but leads to the balanced solution in this special case. Thus $Cost_1(D_{v-1}) = Cost_{min}(G_{v-1})$.

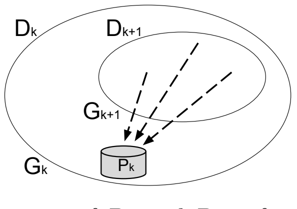


Figure 13: Illustration of D_k and D_{k+1} for induction proof.

Our induction assumption is that the solution for subgraph D_{k+1} is competitive. Namely $Cost_1(D_{k+1}) \leq 2(1+\delta)$ $Cost_{min}(G_{k+1})$. We want to show the solution for D_k is also competitive. Figure 13 illustrates subgraphs D_k and D_{k+1}. Note that subgraph D_k and G_k both have $v{-}k{+}1$ nodes and without loss of generality, these partition nodes are called $P_k, P_{k+1}, \cdots, P_v$. $Cost_{min}(G_k)$ satisfies

$$\begin{aligned}
Cost_{min}(G_k) &\geq \frac{\sum_{j=k}^{v} f(P_j,P_j) + \sum_{k\leq i<j\leq v, e_{i,j}\in G_k} f(P_i,P_j)}{v-k+1} \\
&= \frac{\sum_{j=k}^{v} f(P_j,P_j) + \sum_{j=k}^{v} PW(G_k,P_j)}{2(v-k+1)} \\
&> \frac{\sum_{j=k}^{v} PW(G_k,P_j)}{2(v-k+1)} \\
&\geq \frac{(v-k+1)PW(G_k,P_k)}{2(v-k+1)} \\
&= \frac{1}{2}PW(G_k,P_k).
\end{aligned}$$

Also notice that graph G_{k+1} is a subgraph of G_k, then

$$Cost_{min}(G_k) \geq Cost_{min}(G_{k+1}).$$

Also following the definition of δ and the setting of $Cost(T_k)$ in Stage 1 of two-stage load assignment,

$$Cost(T_k) \leq PW(G_k,P_k)(1+\delta).$$

With the induction assumption and the above three inequalities, the outcome of Stage 1 with respect to D_k satisfies

$$\begin{aligned}
Cost_1(D_k) &= \max\{Cost_1(D_{k+1}), Cost(T_k)\} \\
&\leq \max\{2(1+\delta)Cost_{min}(G_{k+1}), PW(G_k,P_k)(1+\delta)\} \\
&\leq (1+\delta)\max\{2Cost_{min}(G_k), 2Cost_{min}(G_k)\} \\
&= 2(1+\delta)Cost_{min}(G_k).
\end{aligned}$$

Therefore

$$Cost(D) \leq Cost_1(D) = Cost_1(D_1) \leq 2(1+\delta)Cost_{min}(G).$$

□

Theorem 2

PROOF. First we examine the Gantt chart of the schedule from time 0 to PT_q, identifying the total computation and I/O cost, and the idle time. Define the total computation cost as $\pi = \sum_{P_i\in D} f(P_i,P_i) + \sum_{e_{j,i}\in D} f(P_i,P_j)$, where D is the comparison graph generated by two-stage load assignment. Then the total computation and I/O cost is bounded by $\pi(1+\delta)$. Since the scheduling algorithm assigns a task whenever there is an idle core available, the total idle time in all q cores from time 0 to time PT_q is at most $(q{-}1)Cost(D)$. Then

$$\max(Cost(D), \frac{\pi}{q}) \leq PT_q \leq \frac{(q-1)Cost(D) + \pi(1+\delta)}{q}.$$

Given an optimal schedule for similarity graph G on q cores, a comparison graph can be derived. Let $Cost_{opt}(G)$ be the largest task cost in this comparison graph. Notice

$$Cost_{min}(G) \leq Cost_{opt}(G).$$

The optimal solution satisfies

$$\max(Cost_{opt}(G), \frac{\pi}{q}) \leq PT_{opt}.$$

Following Theorem 1,

$$PT_q \leq \frac{q-1}{q}2(1+\delta)Cost_{min}(G) + (1+\delta)PT_{opt}.$$

Thus

$$\frac{PT_q}{PT_{opt}} \leq \frac{q-1}{q}2(1+\delta)+(1+\delta)=(3-\frac{2}{q})(1+\delta).$$

$\square$

Theorem 3

PROOF. Assume that the number of partitions v is an odd number and we show that all tasks formed have equal weights. The optimality for an even number v can be proved similarly.

Since all nodes have the same self-comparison cost, the same cost to compare with others, and the same cost for communication and data I/O, the cost of each task is proportional to the number of incoming edges for the corresponding node in D. We claim that every node at the end of load assignment has $\frac{v-1}{2}$ incoming edges in comparison graph D, namely it compares with $\frac{v-1}{2}$ neighbors.

We prove by contradiction. If some nodes have the number of incoming edges different from $\frac{v-1}{2}$, then some nodes must have more than $\frac{v-1}{2}$ incoming edges while some other nodes must have less than $\frac{v-1}{2}$ edges since the total number of edges is $\frac{v(v-1)}{2}$ for a fully connected graph. Assume the heaviest nodes P_x has more than $\frac{v-1}{2}$ incoming edges, and there exists an incoming edge from node P_y with the number of incoming edges less than or equals to $\frac{v-1}{2}-1$. Figure 14 illustrates an example with contradiction.

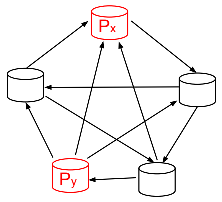


Figure 14: An example for proof by contradiction.

Given all partitions have the equal size, Stage 2 of load assignment should not have stopped since it could reverse the edge between T_x and T_y, causing the decrease of $Cost(T_x)$ while $Cost(T_y)$ does not exceed the new value of $Cost(T_x)$. That is a contradiction.

Thus each task T_i formed fetches from its $\frac{v-1}{2}$ neighbors. Tasks have the same weight, leading to a perfect task distribution among q cores. Without loss of generality, we use $f(P_i, P_i)$, $f(P_i, P_j)$, and $f_c(P_i)$ to represent the cost of self-comparison, inter-partition comparison, and data I/O respectively for all tasks. Then

$$\begin{aligned} PT_q &= \frac{v}{q}(f(P_i,P_i)+f_c(P_i)+\frac{v-1}{2}(f(P_i,P_j)+f_c(P_j))) \\ &\leq \frac{v}{q}(f(P_i,P_i)+\frac{v-1}{2}f(P_i,P_j))(1+\delta). \end{aligned}$$

The above upper bound without factor $1+\delta$ is the lower bound for any schedule including the optimum. Thus the solution derived is within $1+\delta$ of the optimum. $\square$

A. REFERENCES

[1] F. Aiolli. Efficient top-n recommendation for very large scale binary rated datasets. In *Proc. of 7th ACM Conf. on Recommender Systems*, pages 273–280, 2013.

[2] M. Alabduljalil, X. Tang, and T. Yang. Cache-conscious performance optimization for similarity search. In *ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 713–722, 2013.

[3] M. Alabduljalil, X. Tang, and T. Yang. Optimizing parallel algorithms for all pairs similarity search. In *ACM Inter. Conf. on Web Search and Data Mining*, pages 203–212, 2013.

[4] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proc. of VLDB '2006*, pages 918–929.

[5] L. T. I. at Carnegie Mellon University. The clueweb09 dataset, http://boston.lti.cs.cmu.edu/data/clueweb09.

[6] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, a. Ziviani, and N. Ziviani. Analyzing imbalance among homogeneous index servers in a web search system. *Information Processing & Management*, 43(3):592–608, May 2007.

[7] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, 1999.

[8] R. Baraglia, G. D. F. Morales, and C. Lucchese. Document similarity self-join with mapreduce. In *2010 IEEE Inter. Conf. on Data Mining*, pages 731–736.

[9] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proc. of Inter. Conf. on World Wide Web*, WWW '2007, pages 131–140. ACM.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[11] A. Chowdhury, O. Frieder, D. A. Grossman, and M. C. McCabe. Collection statistics for fast duplicate document detection. *ACM Trans. Inf. Syst.*, 20(2):171–191, 2002.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04*, pages 137–150.

[13] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with mapreduce. In *ACL '2008*, pages 265–268.

[14] M. R. Garey and R. L. Grahams. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4:187–200, 1975.

[15] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[16] H. Hajishirzi, W. tau Yih, and A. Kolcz. Adaptive near-duplicate detection via similarity learning. In *Proc. of ACM SIGIR*, pages 419–426, 2010.

[17] N. Jindal and B. Liu. Opinion spam and analysis. In *Proc. of ACM WSDM'2008*, pages 219–230.

[18] E. Kayaaslan, S. Jonassen, and C. Aykanat. A Term-Based Inverted Index Partitioning Model. *ACM Transactions on the Web (TWEB)*, 7(3):1–23, 2013.

[19] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *Proc. of ACM SIGIR'2009*, pages 155–162.

[20] A. Metwally, D. Agrawal, and A. El Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In *Proc. of WWW'2007*, pages 241–250. ACM.

[21] G. D. F. Morales, C. Lucchese, and R. Baraglia. Scaling out all pairs similarity search with mapreduce. In *8th Workshop on Large-Scale Distri. Syst. for Information Retrieval*, 2010.

[22] M. Sahami and T. D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *Proc. of WWW '2006*, pages 377–386. ACM.

[23] N. Shivakumar and H. Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proc. of DL'96*, pages 160–168.

[24] F. Ture, T. Elsayed, and J. Lin. No free lunch: brute force vs. locality-sensitive hashing for cross-lingual pairwise similarity. In *Proc. of SIGIR'2011*, pages 943–952.

[25] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proc. of ACM SIGMOD 2010*, pages 495–506.

[26] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. *Proc. of ACM SIGKDD*, pages 829–837, 2013.

[27] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proc. of WWW 2008*, pages 131–140. ACM.

[28] M. Zaharia, K. Elmeleegy, D. Borthakur, S. Shenker, J. S. Sarma, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of EuroSys*, pages 265–278, 2010.